University of Wollongong

Research Online

Faculty of Engineering and Information
Sciences - Papers: Part A

Faculty of Engineering and Information
Sciences

2013

Automatic BDI plan recognition from process execution logs and effect logs

Hongyun Xu

University of Wollongong

Bastin Tony Roy Savarimuthu

University of Otago, bostin@uow.edu.au

Aditya Ghose

University of Wollongong, aditya@uow.edu.au

Evan Morrison

University of Wollongong, edm92@uowmail.edu.au

Qiyang Cao

Donghua University

See next page for additional authors

Follow this and additional works at: <https://ro.uow.edu.au/eispapers>

Part of the [Engineering Commons](#), and the [Science and Technology Studies Commons](#)

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

Automatic BDI plan recognition from process execution logs and effect logs

Abstract

Agent applications are often viewed as unduly expensive to develop and maintain in commercial contexts. Organizations often settle for less sophisticated and more traditional software in place of agent technology because of (often misplaced) fears about the development and maintenance costs of agent technology, and the often mistaken perception that traditional software offers better returns on investment. This paper aims to redress this by developing a plan recognition framework for agent program learning, where behavior logs of legacy applications (or even manually executed processes) are mined to extract a 'draft' version of agent code that could eventually replace these applications or processes. We develop, implement and evaluate techniques for inferring agent plans from behavior logs, with both positive and negative examples. After obtaining the plans, we resort to an effect log to identify the context (i.e. precondition) for each plan. The experimental results show that our framework generates a first draft of an agent program (i.e. the code) which can then be modified as required by a developer.

Keywords

logs, effect, plan, bdi, recognition, automatic, process, execution

Disciplines

Engineering | Science and Technology Studies

Publication Details

Xu, H., Savarimuthu, B., Ghose, A., Morrison, E., Cao, Q. & Shi, Y. (2013). Automatic BDI plan recognition from process execution logs and effect logs. Lecture Notes in Computer Science, 8245 274-291.

Authors

Hongyun Xu, Bastin Tony Roy Savarimuthu, Aditya Ghose, Evan Morrison, Qiying Cao, and Youqun Shi

Automatic BDI Plan Recognition from Process Execution Logs and Effect Logs

Hongyun Xu^{1,3}, Bastin Tony Roy Savarimuthu², Aditya Ghose³, Evan Morrison³,
Qiyong Cao¹, Youqun Shi¹

¹School of Information Science and Technology,
Donghua University, Shanghai, 201620, P.R. China

²Department of Information Science,
University of Otago, PO Box 56, Dunedin, New Zealand

³College of Computer Science and Software Engineering,
University of Wollongong, Wollongong, NSW, 2522, Australia
xjyc2007@gmail.com, tonyr@infoscience.otago.ac.nz,
aditya@uow.edu.au, caoqiying, yqshi@dhhu.edu.cn

Abstract. Agent applications are often viewed as unduly expensive to develop and maintain in commercial contexts. Organizations often settle for less sophisticated and more traditional software in place of agent technology because of (often misplaced) fears about the development and maintenance costs of agent technology, and the often mistaken perception that traditional software offers better returns on investment. This paper aims to redress this by developing a plan recognition framework for *agent program learning*, where behavior logs of legacy applications (or even manually executed processes) are mined to extract a ‘draft’ version of agent code that could eventually replace these applications or processes. We develop, implement and evaluate techniques for inferring agent plans from behavior logs, with both positive and negative examples. After obtaining the plans, we resort to an effect log to identify the context (i.e. precondition) for each plan. The experimental results show that our framework generates a first draft of an agent program (i.e. the code) which can then be modified as required by a developer.

1 Introduction

Agent applications are often viewed as unduly expensive to develop and maintain in commercial contexts. Even in organizations where there is recognition that the agent technology has benefits, people often settle for less sophisticated, and more traditional software in place of agent technology because of (often misplaced) fears about the development and maintenance costs of agent technology, and the (often mistaken) perception that traditional software offers better return on investment [8]. This paper posits that *agent program learning* offers a solution to this problem. We define the general *agent program learning* problem as follows:

- *Given*: A *behavior log* describing the behavior of a system over a given audit period, and an *effect log*¹ describing the states of the system at some point in time.
- *Determine*: An *agent program* such that if that agent program were executed with the same set of inputs from the environment, the original behavior log would be obtained (with the exception of failure instances, to be discussed below, which would be avoided).

An agent program learning system can lead to significant improvements in programmer productivity. Instead of having to develop an agent program from scratch, which is an expensive and time-consuming proposition, especially in light of the well-known *knowledge acquisition problem* [3,13], an agent programmer would be provided with an initial, ‘draft’ program, which the programmer could edit with far less effort relative to writing a program from scratch to obtain a complete and correct agent program. Agent technology is often deployed to ‘upgrade’ legacy applications [8]. The behavior logs required for an agent program learning system could thus be obtained by auditing the behavior of the legacy system.

A typical BDI agent program consists of three components [9]: (1) A set of *beliefs*, which may be dynamically generated by sensor inputs. (2) A set of *plans*, where each plan contains a triggering event, context conditions and plan body (a set of action sequences). (3) A set of *goals* that an agent wants to achieve. Each goal can be achieved by executing plans in the plan library. The goals involved are related to plans recognized and are given a unique label (e.g. goal *g1* can be achieved by executing plan *p1*).

Thus, the *general agent program learning system* which is the overarching direction of our work, should have modules to support the following: (1) plan recognition; (2) goal recognition; (3) preference recognition². In this paper, we only focus on *plan recognition* for learning a BDI agent program, as exemplified by the BDI agent programming language AgentSpeak(L) [9]. In other words, the plans of an agent program that are automatically created by our system will conform to Jason’s AgentSpeak specification [1]. The plan recognition activity comprises of two parts: a) plan body recognition and b) context recognition. Our contributions to the field of the automatic plan recognition are two-fold: i) we show how the plans are inferred both from positive and negative examples, i.e. behavior logs containing both successful and failed actions; ii) we identify contexts using effect logs to form the precondition for a plan.

This paper is organized as follows. Section 2 introduces the related work on plan recognition. Section 3 provides an overview of the plan recognition framework. Section 4 describes how the plan body recognition works using an example from the banking domain. Both positive and negative behavior logs of process executions are used as inputs for inferring plans. Section 5 presents the algorithm used for context

¹ An effect log contains the resultant states of a program as it continues executing different steps. In contrast, the behavior log contains details about actions that succeeded or failed. In other words, the behavior log captures actions that are executed and the effect log captures the state of the system.

² Each agent can have a set of preferences and these preferences contribute to the selection of an appropriate plan from a set of available plans [2].

recognition. The algorithm uses the positive logs, effects logs and the plan body inferred in the previous step as inputs to infer contexts. Section 6 presents the pilot experiment we have conducted and discusses its results. Section 7 draws conclusions from the results and discusses future research directions.

2 Preliminaries and the Overview of the Plan Recognition Framework

2.1 WF-Nets

Petri nets [7] are a graphical and mathematical modeling notation that allows users to describe business processes. Formally, a Petri net is a tuple (P, T, F) , where P is a set of places, T is a set of transitions ($P \cap T = \emptyset$), and $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs between the places and transitions. A Petri net which models the control-flow dimension of a workflow, is called a WF-net. Let $N = (P, T, F)$ be a Petri net, N is a WF-net if and only if it satisfies the following three requirements. (1) It has only one input place $i \in P$, such that $\bullet i = \emptyset$ ³. (2) It has only one output place $o \in P$, such that $o \bullet = \emptyset$ ⁴. (3) If a new transition t' is added to N , $T \cup \{t'\}$, and t' is used for connecting the output place to the input place, $F \cup \{(o, t'), (t', i)\}$, such that the new net is strongly connected, $N' = (P, T \cup \{t'\}, F \cup \{(o, t'), (t', i)\})$. The WF-net that we use in this work is a structured WF-net (SWF-net) [13]. A sample WF-net is given in Figure 2.

2.2 Norms Identification

Norms are the societal rules that govern the prescription and proscription of certain behavior. In multi-agent systems, norms are viewed mostly as constraints on the actions that an agent can perform. We focus here on two types of norms that are obligation norms and prohibition norms. For obligation norms, it means that a certain action must happen after a specified action. For prohibition norms, it means that a certain action or some actions are prohibited to happen before or after a correct action.

Savarimuthu et al. have proposed two algorithms, prohibition norm identification (PNI) [10] and obligation norm identification (ONI) [11] respectively. The starting point for ONI and PNI is sanction recognition. Once sanctions are recognized, the reasons for these sanctions are investigated (i.e. norm violations are reasons for sanctions to occur). PNI algorithm [11] identifies those actions that occur 100% of the time before the sanctions. ONI algorithm [10] identifies those actions that fail to occur whenever a sanction occurs. In order to identify those actions it compares two lists of action sequences, one containing missing actions (followed by sanctions) and the other containing the expected actions (without sanctions). On comparing these lists,

³ The expression $\bullet i = \emptyset$ denotes that there are no incoming edges to the input place i (the dot here represents a set of transitions).

⁴ The expression $o \bullet = \emptyset$ denotes that there are no outgoing edges from the output place o (the dot here represents a set of places).

the algorithm identifies missing actions. These algorithms were originally proposed to be used in artificial agent societies where interacting avatars can automatically learn norms based on observation of actions. For example, an agent that litters a park might be sanctioned by another agent. An observer, based on actions observed can infer that littering is prohibited. Also, an agent that does not tip (a violation of an obligation) might be sanctioned. PNI and ONI algorithms identify two norms, i.e., prohibit(litter) and obliged(tip)) respectively.

The PNI and ONI algorithms can not only identify norms but also preconditions and post conditions of norms. This is useful in our context, because if we view agents as entities being norm aware (e.g. action x is prohibited and action y is obliged), then they need to make plans to adhere to norms. Hence, the precondition and post-condition can be viewed as parts of a plan for achieving a goal (normative goal in this case). Failures in business processes can happen because of many reasons such as equipment malfunction, actions happening out of order or a required action not being performed. In this work, we assume failures happen because of the latter two and the failures are recorded in behavior logs (i.e. negative examples). We use the negative examples to infer plans.

2.3 Framework Overview

The goal of this work is to describe how a first draft of an agent’s plans can be automatically recognized from actions recorded in the business process execution log (i.e. a behavior log) and the effect log. The process of plan recognition is shown in Figure 1. A plan consists of two main aspects, a plan body and a context (i.e. precondition that results in the execution of a plan body when it evaluates to true). These two aspects of plan recognition are carried out by two modules namely the plan body recognizer and context recognizer.

Fig. 1: Overview of the plan recognition framework

A typical behavior log contains information about both successful process executions and failures. In our work, from the log containing successful executions, we generate a workflow diagram (i.e., a WF-net) using the process mining tool, ProM [14]. Then, we demonstrate how a plan body recognizer (a set of transformation rules and

algorithms) can be used to infer BDI plans (without context) from the WF-net in Section 3. Then, the context recognizer is used to identify the context which takes an effect log, the behavior log, and the plan body that was identified in the previous step (details provided in Section 4). The plans obtained from the positive examples are hierarchical in nature (nested plans with successful actions). Also, based on failure sequences, we demonstrate how a data mining based approach previously used for norm extraction can be used to identify sequential normative actions (obligations and prohibitions).

3 Plan Body Recognition

In this section, we first describe a motivating example that is used throughout the paper. Then we present the plan body recognition, first from handling positive behavior logs containing no failure information, and then from negative behavior logs containing failure information.

3.1 Motivating Example

A typical banking system contains many business applications. We refer to two of these business applications throughout the paper. They are the processes in a banking system for *loan applications* and *money transfers*, respectively. Assume that a banking system (or an agent) receives loan applications which are handled according to the type of loan (e.g. personal and business loans). If a loan application is for a personal loan, personal information such as credit history and the risk (e.g. high, medium, low), will be evaluated and then a decision will be made. At the end of the process, the applicant will be informed of the acceptance or rejection. If an application involves a business loan, the bank will check whether the information provided is correct and also check other relevant sources in order to assess the credibility of the business entity (e.g. whether it is registered member of the chamber of commerce). The bank will also audit and evaluate the business's assets. Then it will make a decision and also notify the business. If the banking system receives a money transfer application during the handling process for loan applications, the account of the applicant will be checked to see if the money in the account is enough to be transferred. Then it will check whether the target account specified in the application exists. Then, the money will be transferred by the banking system. The examples includes a set of actions as follows: *receive_loan_application*, *check_personal_loan_application*, *check_business_loan_application*, *audit_assets*, *check_credit*, *check_risk*, *evaluate_personal_loan_application*, *inform_loan_applicant*, *receive_money_transfer_application*, *check_applicant_account*, *check_target_account* and *evaluate_money_transfer_application*.

3.2 Handling Positive Behavior Logs

In this sub-section, we describe how a behavior model is constructed from the actions available in a behavior log, and then present the transformation rules and the algorithms that transform a WF-net to BDI plans.

Behavior Model Construction - As shown in Table 1, there are five cases in a positive behavior log from a banking system where each action in the behavior log has a timestamp indicating the starting time of that action. For example, the actions associated with case 1 are $\langle receive_loan_application, check_personal_loan_application, check_credit, check_risk, evaluate_personal_loan_application, inform_loan_applicant \rangle$. The actions appear in the order they were executed. The actions logged for case 2 are $\langle receive_loan_application, check_personal_loan_application, check_risk, check_credit, evaluate_personal_loan_application, inform_loan_applicant \rangle$, and actions logged for cases 3 and 5 are $\langle receive_loan_application, check_business_loan_application, audit_assets, inform_loan_applicant \rangle$, and the actions executed as a part of case 4 are $\langle receive_money_transfer_application, check_applicant_account, check_target_account$ and $evaluate_money_transfer_application \rangle$.

Table 1: Sequential Behavior Log

Case ID	Action ID	TimeStamps(yyyy/mm/dd hh:mm)
case 1	<i>receive_loan_application</i> (rla)	2011/06/19 08:53 (t_1)
case 1	<i>check_personal_loan_application</i> (cpla)	2011/06/19 11:00 (t_2)
case 2	<i>receive_loan_application</i> (rla)	2011/06/19 11:10 (t_3)
case 1	<i>check_credit</i> (cc)	2011/06/19 12:38 (t_4)
case 3	<i>receive_loan_application</i> (rla)	2011/06/19 12:40 (t_5)
case 2	<i>check_personal_loan_application</i> (cpla)	2011/06/19 12:51 (t_6)
case 4	<i>receive_money_transfer_application</i> (rmta)	2011/06/19 13:28 (t_7)
case 1	<i>check_risk</i> (cr)	2011/06/19 13:56 (t_8)
case 5	<i>receive_loan_application</i> (rla)	2011/06/19 14:01 (t_9)
case 1	<i>evaluate_personal_loan_application</i> (epla)	2011/06/19 14:53 (t_{10})
case 3	<i>check_business_loan_application</i> (cblla)	2011/06/19 15:24 (t_{11})
case 4	<i>check_applicant_account</i> (caa)	2011/06/19 15:32 (t_{12})
case 1	<i>inform_loan_applicant</i> (ila)	2011/06/19 15:41 (t_{13})
case 3	<i>audit_assets</i> (aa)	2011/06/19 15:59 (t_{14})
case 5	<i>check_business_loan_application</i> (cblla)	2011/06/19 16:04 (t_{15})
case 4	<i>check_target_account</i> (cta)	2011/06/19 16:13 (t_{16})
case 2	<i>check_risk</i> (cr)	2011/06/19 16:25 (t_{17})
case 5	<i>audit_assets</i> (aa)	2011/06/19 16:30 (t_{18})
case 2	<i>check_credit</i> (cc)	2011/06/19 16:48 (t_{19})
case 3	<i>inform_loan_applicant</i> (ila)	2011/06/19 16:55 (t_{20})
case 4	<i>evaluate_money_transfer_application</i> (emta)	2011/06/19 17:08 (t_{21})
case 2	<i>evaluate_personal_loan_application</i> (epla)	2011/06/19 17:10 (t_{22})
case 5	<i>inform_loan_applicant</i> (ila)	2011/06/19 17:17 (t_{23})
case 2	<i>inform_loan_applicant</i> (ila)	2011/06/19 17:34 (t_{24})

The plan recognition framework uses the ProM tool to construct a behavior model, i.e., a WF-net that describes the behavior of a system based on the actions recorded in the behavior log. ProM enables the extraction of information from event logs, and it supports several process mining techniques in the form of plug-ins, such as

the Alpha-algorithm [13]. We exploit the Alpha-algorithm in ProM to automatically generate a WF-net from the log.

The WF-net shown in Figure 2, is constructed from the log given in Table 1 using the Alpha-algorithm in ProM. The start and end transitions as endpoints are added automatically when there is no common start action (or actions) and no common end action (or actions) among cases in the behavior logs. This is required to tie-together distinct set of processes a system can execute (e.g. a business system can run ten different processes at any point of time) and a WF-net is a representation of all these different processes. The other transitions in the diagram are the actions that an agent must perform. The places in the WF-net are labeled $p_1, p_2, \dots, p_{14}$ for easier referencing.

Fig. 2: A WF-net generated based on a behavior log

Transformation Rules and Algorithms - WF-nets are commonly used to represent a process which is composed of various applications (e.g., different business applications in the context of the banking example). The business process when executed will produce different outputs depending upon the inputs given to the process (e.g., the path taken for evaluating a personal loan will be different from the path taken for evaluating money transfer application). That is, the business process will produce different results (i.e. outputs) depending on the triggering events (i.e., inputs). We argue that the behavior exhibited by the traditional system could be viewed as the behavior exhibited by an agent that follows some plans (i.e., plans for different goals are different). Different triggering events could be handled by distinct plans. In Figure 2, the condition in place p_2 could be triggered by different external events such as submission of personal loan application and money transfer application. Hence, we argue that, it is reasonable to view the number of branches emanating p_2 as the number of plans for different goals. That is, the paths that originate from p_2 can be different, where each path represents a different goal (e.g., the plan involving actions from case 4, represents the goal of transferring money from one account to another). In our work, each inferred plan is a BDI plan, consisting of a sequence of actions and/or subgoals. Before presenting the algorithms for generating plans for a goal, we first present the *types of sub WF-nets* and the related *rules* for inferring plans from WF-nets.

A sub WF-net starts at a node N and ends at node N' where $N, N' \in P \cup T$ (as per usual convention $\bullet N = \emptyset$ and $N' \bullet = \emptyset$). The number of branches deviating at N is greater than 1, i.e. $|N \bullet| > 1$. At N' all of the branches join together, i.e., $|\bullet N'| > 1$. A sub WF-net is called *WNOP* (WF-net for Optional Plans) iff $N, N' \in P$, highlighted using solid boxes in Figure 2. And, a sub WF-net is called *WNPP* (WF-net for Parallel Plans) iff $N, N' \in T$, highlighted using the dotted box in Figure 2. The *transformation rules* for inferring plans from WF-nets are as follows.

1. The number of top-level plans for achieving different goals are determined by the number of branches emanating the node (the place), that is located next to the start transition.
2. Each transition is viewed as an action in a plan, in other words, a transition node has the same label as an action.
3. Each sub WF-net is considered as a subgoal in a plan.
4. If a subgoal is of *WNOP* type, the number of possible plans achieving the subgoal is determined by the number of branches emanating the start node in the sub WF-net, and these obtained plans are alternative plans, of which only one will be chosen for execution at run-time.
5. If a subgoal is *WNPP* type, the number of possible plans relies on the number of branches emanating the start node in the sub WF-net, and these plans can be executed in parallel.

Algorithm 1 Recognize plans for one goal

Input: wf, i.e., a WF-net; currentNode

Output: planList, i.e., BDI agent-oriented plans

```

1: while currentNode  $\neq$  wf.endNode do
2:   if currentNode.type is transition AND currentNode.type  $\neq$  wf.startNode then
3:     put currentNode.name in a plan
4:   end if
5:   if count(currentNode.outgoingEdges)= 1 then
6:     currentNode $\leftarrow$ currentNode.nextNode
7:   else if count(currentNode.outgoingEdges)>1 then
8:     generate a new subgoal in a plan
9:     if currentNode.type is place then
10:      planList $\leftarrow$  obtain plans for WNOP goal
11:     else if currentNode.type is transition then
12:      planList $\leftarrow$  obtain plans for WNPP goal
13:     end if
14:   end if
15: end while
16: planList  $\leftarrow$  planList  $\cup$  {plan}
17: return planList

```

Algorithm 1, presents the process of inferring plans for one goal. If there are multiple goals, the algorithm will be used iteratively (which is the case in our work

for the WF-net given in Figure 1). Also, note that top-level goals (first iteration goals) are represented as *goal1* and *goal2*. The next level goals are represented as *subgoal1*, *subgoal2* and so on. The WF-net is encoded as a graph, and the node after start transition in the WF-net is viewed as *currentNode*. Each node has its type, i.e., place and transition, and its name. If the node type is transition, its name will be viewed as an action label, which will be added in a plan, following rule (2). If there exists a sub WF-net, a new subgoal is generated in the plan, following rule (3). If a subgoal is *WNOP* type, the number of plans for the subgoal depends on the number of branches emanating the start node of the sub WF-net, as described in rule (4). Each branch constructs a new plan for the subgoal, recursively using the Algorithm 1. Likewise, for the *WNPP* type of subgoal, the number of plans for this subgoal depends on the number of branches emanating from start node of the sub WF-net, as presented in rule (5). There is no parallel construct in AgentSpeak(L) [1], hence we handle the *WNPP* type of subgoal using the interleaved actions among the paths to obtain all possible plans. In the running example, two actions E and F can be executed in parallel. It means that, there are two possible ways in which these actions could have been executed, E followed by F or F followed by E.

The results of using Algorithm 1 when the WF-net shown in Figure 2 is given as the input (recursively for each goal) are given in Table 2. There are six plans in total (p_0 to p_5). We can observe that there are two top-level goals achieved by these plans, i.e., *goal1* and *goal2* (rule 1). The first goal, *goal1* can be achieved directly using plan p_0 , while the second goal *goal2* can be achieved using p_1 that has a subgoal *subgoal1*. There are two alternative ways to achieve this subgoal (either using plan p_2 or using plan p_3). Note that plans p_0 and p_1 are of *WNOP* type. So are plans p_2 and p_3 that are used to achieve *subgoal1*. On the other hand plans p_4 and p_5 are of *WNPP* type. They are used to achieve *subgoal2*. Note that the preconditions for each of the plans is set to true. Identifying preconditions of these plans is discussed in Section 5.

Table 2: Results of Plan Body Recognition

@ p_5	+!subgoal2: true $\leftarrow$ <i>check_credit</i> ; <i>check_risk</i> .
@ p_4	+!subgoal2: true $\leftarrow$ <i>check_risk</i> ; <i>check_credit</i> .
@ p_3	+!subgoal1: true $\leftarrow$ <i>receive_personal_loan_application</i> ; !subgoal2; <i>evaluate_personal_loan_application</i> .
@ p_2	+!subgoal1: true $\leftarrow$ <i>check_business_loan_application</i> ; <i>audit_assets</i> .
@ p_1	+!goal2: true $\leftarrow$ <i>receive_loan_application</i> ; !subgoal1; <i>inform_loan_applicant</i> .
@ p_0	+!goal1: true $\leftarrow$ <i>receive_money_transfer_application</i> ; <i>check_applicant_account</i> ; <i>check_target_account</i> ; <i>evaluate_money_transfer_application</i> .

3.3 Handling Negative Behavior Logs

In this sub-section, we discuss how plans are extracted from a behavior log that contains failure information. Failures could be caused when obligated actions do not occur or when prohibited actions are performed. We acknowledge that there could be other reasons for failures such as a printer failing because of power failure. We do not model those because those type of failures are explicit failures and the reasons are

known to the agent (i.e., power failure is the reason for failed printing job). Obligated and prohibited actions can be identified using ONI and PNI algorithms.

Failures in business process executions are logged in behavior logs (e.g. failure of a task due to a mechanical error or human error). These failures are similar to sanctions. When a sanction happens, a special event (\$) gets recorded in the norm identification framework. Similarly, in this work, when a failure happens a special event (⊗) will be recorded. When these failures happen, the reasons for the failures can be investigated. In our work we assume failures happen either because a prohibited action is performed (i.e. the action that does not fit in a sequence, such as sending rejection notification before decision is made) or an obliged action does not happen (e.g. credit check is not performed before a decision can be made). We use slightly modified versions of PNI and ONI to identify the reasons for failures (prohibited and obliged actions respectively). The modifications made are two fold. First, we eliminate the norm verification stage (where an agent asks another agent to verify whether a norm holds) because we only consider an action to be prohibited by setting the threshold for norm identification to be 100% (i.e. only those actions that have the probability of 1 to be causing violations are considered). Second, we create two lists to handle the two types of norms separately (details discussed in Algorithm 2).

Algorithm 2, describes the process for identifying plan sequences from a behavior log with failure using these two norm learning algorithms. First, the correct entries without failure information are stored in a correct list, and are also removed from the behavior log (lines 4-9). Second, the entries with failure information are classified into two lists, prohibition list and obligation list. If a failed sequence (i.e. a case) has a matching super sequence(s) in the correct list, then that sequence will be added to the obligation list. Assume actions a, b and c were supposed to happen in sequence. Let us assume that the observed sequence is ac , resulting in a failure. Here action b is the obligated action. By identifying the supersequence of ac (which is abc), we are able to identify the obliged action (i.e. b). Otherwise, it will be added to a prohibition list (lines 10-17). Third, the ONI and PNI algorithms are invoked (lines 18, 19). Given the correct list and the obligation list as inputs to the ONI algorithm, it produces plan sequences as outputs. Next, the PNI algorithm is invoked. Given the correct list and the prohibition list as inputs, it produces plan sequences as output.

To demonstrate how Algorithm 2 works, a sample log with failure information is given in Table 3. Note that ⊗ is used to indicate failures in the behavior log. If there is a failure, then other steps of the process are not executed. The algorithm first creates three empty lists (lines 1-3), the correct list, prohibition list and the obligation list. The correct list is then populated with four entries: (rla, cpla, cc, cr, epla, ila), (rla, cpla, cr, cc, epla, ila), (rla, cbla, aa, ila) and (rmta, caa, cta, emta). Then the algorithm populates the obligation list which contains the following entries: (rla, cpla, cc, cr, ila, ⊗), (cpla, cr, cc, ila, ⊗), (rla, cbla, ila, ⊗) and (rla, cpla, cc, epla, ⊗). The prohibition list contains the following entries: (rmta, Z, ⊗), (rmta, caa, cta, Y, ⊗), (rmta, caa, rmta, ⊗) and (rla, cbla, aa, L, ⊗). Then, the ONI and PNI are executed in sequence to identify plans. The results of using the norm learning mechanisms are shown in the last column of Table 3.

Algorithm 2 Identifying plans from negative behavior logs

Input: bl, i.e., a behavior log containing failures

Output: ps, i.e., plan sequences with obligated actions or prohibited actions

```
1: Let correctList  $\leftarrow \emptyset$   $\triangleright$  entries without  $\otimes$ 
2: Let prohibitionList  $\leftarrow \emptyset$   $\triangleright$  entries with prohibited actions
3: Let obligationList  $\leftarrow \emptyset$   $\triangleright$  entries with missing actions
4: for each entry E  $\in$  bl do
5:   if E contains no  $\otimes$  then
6:     Add E to correctList
7:     Remove E from bl
8:   end if
9: end for
10: for each entry E  $\in$  bl do
11:   remove  $\otimes$  in E
12:   if E has super sequence(s) in correctList then
13:     Add E to obligationList
14:   else
15:     Add E to prohibitionList
16:   end if
17: end for
18: ps  $\leftarrow$  ONI Algorithm(correctList, obligationList)
19: ps  $\leftarrow$  PNI Algorithm(correctList, prohibitionList)
20: return ps
```

There are two benefits of the outputs obtained from the norm learning mechanisms. First, the outputs are the correct sequences (i.e. corrected failure sequences). For example, (rla, cpla, cc, cr, ila, $\otimes$) has been corrected to (rla, cpla, cc, cr, epla, ila). So, the first benefit of norm learning mechanisms is their ability to correct erroneous sequences. Second, the output produced by the algorithms can be viewed as a plan which contains a precondition, an obliged/prohibited action (or actions) and a postcondition. For example, the failure corrected version of (rla, cpla, cc, cr, ila,) is (rla, cpla, cc, cr, Obligated(epla), ila). The precondition of the norm Obligated (epla) is the occurrence of actions rla, cpla, cc, cr, and the the post condition is the occurrence of action ila. A prohibition sequence only contains a precondition and a normative action because there could be many different post conditions depending upon which action should have occurred in the place of the prohibited action.

In the future, agents in our framework can use the results generated to create plans that ensure those already generated plans do not violate the normative action(s) (e.g. none of the existing plans should execute action L after executing actions rla, cpla and aa (entry 12 in Table 3)). This can be particularly valuable when the human programmer modifies the first draft of agent code generated by our framework to suit application needs (i.e. have plans to capture errors accidentally introduced by the humans that could potentially violate the norms). We note that some results obtained are incomplete (e.g. (rla, cpla, cc, epla, $\otimes$)) is corrected to be (rla, cpla, cc, cr, epla, ila) or (rla, cpla, cr, cc, epla, ila). The complete sequence must be (rla, cpla, cc, cr, epla, ila) and (rla, cpla, cr, cc, epla, ila). However, we address this problem

by identifying supersequences of the results in the correct list to produce the correct and complete sequence.

Table 3: A Log with Failure Information

Entry ID	Sequence of Actions	Results from Algorithm 2
1	(rla, cpla, cc, cr, epla, ila)	
2	(rla, cpla, cr, cc, epla, ila)	
3	(rla, cbla, aa, ila)	
4	(rla, cpla, cc, cr, ila, $\otimes$)	(rla, cpla, cc, cr, Obliged(epla), ila)
5	(rla, cpla, cr, cc, $\otimes$, ila)	(rla, cpla, cr, cc, Obliged(epla), ila)
6	(rla, cbla, ila, $\otimes$)	(rla, cbla, Obliged(aa), ila)
7	(rla, cpla, cc, epla, $\otimes$)	(rla, cpla, Obliged(cr), cc, epla) (rla, cpla, cc, Obliged(cr), epla)
8	(rmta, caa, cta, emta)	
9	(rmta, Z, $\otimes$)	(rmta, prohibited(Z))
10	(rmta, caa, cta, Y, $\otimes$)	(rmta, caa, cta, prohibited(Y))
11	(rmta, caa, rmta, $\otimes$)	(rmta, caa, prohibited(rmta))
12	(rla, cbla, aa, L, $\otimes$)	(rla, cbla, aa, prohibited(L))

Table 4: Sample effect log

Time	States	Time	States	Time	States	Time	States
t_1	$c3 \wedge c4$	t_7	$c15$	t_{13}	$c11$	t_{19}	$c7 \wedge c8$
t_2	$c5 \wedge c6$	t_8	$c7 \wedge c8$	t_{14}	$c13 \wedge c14$	t_{20}	$c11$
t_3	$c3 \wedge c4$	t_9	$c3 \wedge \neg c4$	t_{15}	$c12 \wedge \neg c20$	t_{21}	$c18$
t_4	$c1 \wedge c7$	t_{10}	$c9 \wedge c10$	t_{16}	$c17$	t_{22}	$c9 \wedge c10$
t_5	$c3 \wedge \neg c4$	t_{11}	$c12 \wedge c20$	t_{17}	$c1 \wedge c7$	t_{23}	$c11$
t_6	$c5 \wedge c6$	t_{12}	$c16$	t_{18}	$c13 \wedge c14$	t_{24}	$c11$

4 Context Recognition

In Section 4, we described how plans are recognized. However, the preconditions of all the plans were true. Recognizing preconditions is a key part of plan recognition. We note that preconditions cannot be inferred from the behavior log alone. So, we assume, in addition to behavior log we also have the effect log. Behavior log contains actions that were executed and effect log contains the state of the system. For example, the state of the system can contain information such as the loan application of customer 1 was in the *pending allocation* state at time t_1 and the state was changed to *assigned to risk analyst* at t_2 . By using effect log in conjunction with the behavior log and the plan body of the all the recognized plans, we demonstrate how preconditions for the plans can be identified.

We assume that the sample effect log given in Table 4 can be obtained during the execution of a traditional system. We model states using propositional logic (states represented as a conjunction of propositions). We also assume that timestamps of these states are recorded. For example, at timestamp t_1 , c_3 and c_4 hold (assume c_3 is *application received* and c_4 is *data verified for completeness*).

Algorithm 3 Mining preconditions in plans

Input: a) sblog - a behavior log; b) elog - an effect log; c) plans resulting from plan body recognition

Output: plans with context

```

1: for each plan  $\in$  plans do
2:   planTraces  $\leftarrow \emptyset$ 
3:   stateArr  $\leftarrow \emptyset$ 
4:   if plan.planbody has subgoal then
5:     planTraces  $\leftarrow$  obtain all possible plan traces
6:   else
7:     planTraces  $\leftarrow$  plan.planbody
8:   end if
9:   for each trace  $\in$  planTraces do
10:    for each case  $\in$  sblog do
11:      if case contains the trace then
12:         $T_a \leftarrow$  obtain the timeStamp of first action of the plan
13:        if elog contains  $T_a$  then
14:          stateArr  $\leftarrow$  states at  $T_a$  should be adding to stateArr
15:        end if
16:      end if
17:    end for
18:  end for
19:  if stateArr.size  $> 1$  then
20:    plan.context  $\leftarrow$  compute common proposition
21:  else
22:    plan.context  $\leftarrow$  stateArr
23:  end if
24: end for
25: return plans

```

Algorithm 3 presents how preconditions of plans are recognized. The input to the algorithm are a) the behavior log, b) the effect log and c) plan body of all the recognized plans. The algorithm contains three main stages.

Stage 1 (lines 4-8) - For each plan, if it has subgoals, all the possible action sequences for subgoals are identified. This is done by unfolding subgoals recursively to the inner most plan with subgoals. The identified set of these action sequences will be stored in an array called *planTrace*. Each entry in the *planTrace* is a possible execution sequence of a plan (i.e. actions that would be executed if a plan were to be invoked). Note that there could be more than one plan trace for a plan because a subgoal of a plan produce different results. If a plan has no subgoals, the action

sequence itself will be stored in *planTrace* as a plan trace. For example, let us consider plan p_5 which does not have any subgoal. In this case, the *planTrace* contains actions *ef*. However, for plan p_3 that contains a subgoal in its plan body, there will be two plan traces (*befg* and *bfeg*) since subgoal2 can be realized in two different ways.

Stage 2 (lines 9-18) - For each plan trace in *planTrace*, if there exists a case in the behavior log, that contains either the same sequence of the plan trace, or the supersequence of the plan trace, the timestamp of the first action in plan trace is stored in T_a (a variable), and the entry corresponding to T_a in the effect log is stored in an array called *stateArr*. Note that there could more than one one result.

For plan p_5 , the *planTrace* contains the action sequence *ef* as described above. There exists only case (case 1) containing the supersequence of *ef* which is *abefgi*. Then we consider the first action of *ef*, i.e., action *e*, and obtain its timestamp in case 1, which is t_4 . So, the context for p_5 , is the entry corresponding to t_4 in the effect log, which is $c_1 \& c_7$ (which gets stored in *stateArr*). For plan p_3 , there are two plan traces *befg* and *bfeg* as described above. For the plan trace *befg*, case 1 has its supersequence, *abefg*. We obtain the timestamp corresponding to the first action *b* which is t_2 . The entry corresponding to t_2 in the effect log is $c_5 \& c_6$ which is stored in *stateArr*. Similarly, case 2 contains the supersequence of *bfeg* which is *abfeg*. The timestamp corresponding to the first action *b* is identified which is t_6 . The entry $c_5 \& c_6$ in effect log at t_6 is obtained. Note that for plan p_3 , there are two entries in *stateArr* and both these entries have the same propositions (c_5 and c_6).

Stage 3 (lines 19-23) - For a given plan, if the *stateArr* has only one entry, then the entry is the precondition of the plan. If there are more than one entry, then the propositions that are common to these entries will be computed to be the context. For example, for plan p_3 , there were two entries in the *stateArr*. Common propositions among these two entries are chosen as the preconditions for p_3 which are c_5 & c_6 in this case. For plan p_5 , there is only one entry in the *stateArr* ($c_1 \& c_7$) which becomes its precondition.

5 Experiments and Results

Using the simplified motivating example throughout the paper, the plans resulting from context recognition are shown in Table 5, applying the positive behavior log given in Table 1 and the effect log given in Table 4.

We can see that there are two different top-level goals to achieve for an agent, under different contexts (i.e. *goal1* and *goal2*). The context (i.e. precondition) is the same for the parallel plans, p_4 and p_5 , but is different for the optional plans, p_2 and p_3 . However, only one of the parallel plans should be executed at run-time. Since AgentSpeak(L) does not allow two plans to be executed in parallel, one of these plans is randomly picked for execution with respect to our implementation. Note that the plans p_2 and p_3 for achieving *subgoal1* have different context conditions which mean that only one of them will be executed at run time.

Table 5: Results of Plan Body and Context Recognition

@p ₅	+!subgoal2: c1&c7 $\leftarrow$ <i>check_credit; check_risk.</i>
@p ₄	+!subgoal2: c1&c7 $\leftarrow$ <i>check_risk; check_credit.</i>
@p ₃	+!subgoal1: c6&c5 $\leftarrow$ <i>receive_personal_loan_application; !subgoal2; evaluate_personal_loan_application.</i>
@p ₂	+!subgoal1: c12 $\leftarrow$ <i>check_business_loan_application; audit_assets.</i>
@p ₁	+!goal2: c3 $\leftarrow$ <i>receive_loan_application; !subgoal1; inform_loan_applicant.</i>
@p ₀	+!goal1: c15 $\leftarrow$ <i>receive_money_transfer_application; check_applicant_account; check_target_account; evaluate_money_transfer_application.</i>

In order to evaluate the plans generated by our plan recognition framework, we employed human subjects. A pilot study was conducted with eight participants. A small-sized problem specification (including the functional requirements such as the high level goals of the agent program and the expected behavior in terms of output) was provided to the participants, who are students pursuing their postgraduate research work in computer science. They had to handwrite the BDI agent program to achieve these goals. We divided the participants into two groups of four programmers. For one group (group A), we provided just the specification. For the second group (group B) we provided both the specification and the resulting ‘draft’ agent program code generated by our plan recognition framework. Our results show that the average time for programmers in group B to finish the program is much shorter than that of programmers in group A. On average, programmers in group B finished 12 minutes earlier than group A. The maximum time taken to finish the program in group A was 36 minutes, but in group B it was only 17 minutes. The types of errors made by the groups A and B were different. The errors made by programmers in group A include the wrong ordering of actions, assigning wrong preconditions and assigning wrong actions. Group B on the other hand, made few changes to preconditions (addition and deletion). Some did not make changes to the body of the program since the ‘draft’ version provided was adequate in most cases. Since, group B did not start a program from the scratch, they did not make many mistakes. So, the average number of errors in group B was less than that of programmers in group A. Thus, the initial results obtained from the pilot study is promising. However, extensive studies with complex requirements with large number of participants are required to firmly establish our initial findings.

6 Related Work

Very few work have addressed the problem of automatic recognition of plans in the area of agent-oriented software engineering (e.g. [4]). The work in [4] proposes an incremental plan recognition in an agent programming framework, which is similar to our work. However, our work is distinct from theirs. They focus on the formal model of plan recognition based on situation calculus and the ConGolog agent programming language. In their work, the plans are filtered as more actions are observed based on the existing plan library. In contrast, plans in our work are generated from scratch and they are added to an initially empty plan library. Also, we use a process mining

approach to infer BDI plans from both positive and negative examples in behavior logs produced by a traditional system. Our goal is thus to create a framework that generates plans of an agent program which when executed will produce the same behavior as that of a traditional (legacy) system. This new program is the first cut of the ‘agentified’ version of the traditional program.

A difference of our work when compared to the work of Traverso and Pistore [12] where they convert a OWL-based business process to Hierarchical Task Network (HTN) plans is the ability of our system to derive an agent program just based on observed outputs and effects (i.e. we do not start with a process involving a particular technology, i.e. OWL-S), and our approach is generic (i.e. can involve composition). Meneguzzi and Luck [6] have investigated how context conditions can be derived for plans in AgentSpeak(PL), which is similar to our work. A difference between our work and their work is that the cited work generates new plans using AgentSpeak(PL) when a plan in the plan library fails or if an appropriate plan does not exist. However, in our work we start with an empty plan library and add new plans generated by our plan recognition system. Even though the context conditions derived in our work are also plans in AgentSpeak(L), the process of obtaining the plan is different. Their work uses an action model for context derivation (i.e., the precondition and postcondition of actions are known), while our work requires an effect log for context derivation (i.e., the state of a system is known at different points in time). Also, the work [6] do not consider the identification of the erroneous conditions (i.e. the actions that caused failure) and the possibility of creating plans to handle those failures.

There exist other work where method preconditions are learned using the HTN [5, 15]. In [15], a set of constraints are constructed from observed decomposition trees under partial observations, and then solved by a constraint solver. HTN planning systems are related to BDI agent systems when it comes to know-how information used, that is, learning preconditions for a method amounts to learning context condition of a plan in BDI systems. As opposed to the work in [15], our work utilizes a complete behavior log and an effect log for context derivation. Also, we use propositional logic to represent the state of a system at different timestamps, and the actions are without parameters. The work in [5] learns preconditions from plan traces and HTN structures, where the task decomposition is known a priori (i.e. task dependencies are known), while in our work, we derive task composition according to a Workflow net (WF-net, see Section 2.1) transformation rules (discussed in Section 3.2). They use a candidate elimination method to obtain a set of candidate predicates for preconditions (contextual conditions) of plans. However, in our work, we derive context conditions from effect logs obtained (i.e. resultant state of the system). More importantly, our work encompasses a higher level objective of generating a first draft of a plan library.

7 Conclusions and Future Work

In this paper, we have proposed a novel plan recognition mechanism where BDI-style plans are generated by our plan recognition framework. This plan recognition is a part of a larger scope project which aims to learn agent programs (i.e. automatically

generate a first draft of an agent program) from the behavior log and the effect log produced by a traditional (‘legacy’) system. The two main aspects of our plan recognition framework are the plan body recognition and the context recognition. In order to generate plans, first, a WF-net is generated from a behavior log using ProM. Second, we have proposed a set of transformation rules and a procedure (Algorithm 1), which transforms the WF-net into a set of plans (without context). Then, we demonstrated how the preconditions for the plans can be identified using an effect log (Algorithm 3). It should be noted that we have demonstrated how both positive and negative examples can be used to obtain plans. We leveraged existing norm learning mechanisms [10,11] to infer normative plans (with prohibited and obligated actions) in the context of handling negative examples (Algorithm 2).

We have demonstrated that our plan recognition framework creates a ‘draft’ agent program which can then be extended by a programmer. We have conducted a pilot study with eight participants and the results of the study are encouraging. We believe the work presented here is an important step for BDI-type agent systems development since it shows that agent programs can be developed for existing traditional systems (or at least a draft version of the system can be developed). Especially, these agent programs can be considered as a viable alternative for ‘legacy’ systems that need to be redeveloped in an appropriate language.

The plan recognition framework has some simplifying assumptions. First, business processes with loops have not been considered in this work. That forms the focus of our future work. Second, the propositional logic is used to demonstrate the feasibility of the system. Other more expressive logics could be investigated in the future. Also, parameterized actions and states can be included in the behavior log and effect log respectively. In the future, we plan to evaluate our plan recognition framework with complex applications (complex WF-nets). We believe it is in those complex systems, our framework will offer significant advantages (i.e. reduction of programming time and effort). We will also conduct extensive testing involving substantial number of developers.

References

1. R. H. Bordini, J. F. Hübner, and M. J. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, Ltd, 2007.
2. A. Dasgupta and A. K. Ghose. BDI agents with objectives and preferences. In *Proc. of the international workshop on declarative agent languages and technologies (DALT)*, volume 6619 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2010.
3. J. M. Gómez-Pérez, M. Erdmann, M. Greaves, O. Corcho, and R. Benjamins. A framework and computer system for knowledge-level acquisition, representation, and reasoning with process knowledge. *International Journal of Human-Computer Studies*, 68(10):641 – 668, 2010.
4. A. Goultiaeva and Y. Lespérance. Incremental plan recognition in an agent programming framework. In *Working notes of the American Association for Artificial Intelligence (AAAI) Workshop on Plan, Activity, and Intention Recognition (PAIR)*, 2007.
5. O. Ilghami, D. S. Nau, and D. W. Aha. Learning preconditions for planning from plan traces and HTN structure. *Computational Intelligence*, 21:413, 2005.

6. F. R. Meneguzzi and M. Luck. Leveraging new plans in AgentSpeak(PL). In *Proc. of the international workshop on declarative agent languages and technologies (DALT)*, volume 5397 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2008.
7. T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the Institute of Electrical and Electronics Engineers (IEEE)*, volume 77, pages 541–580, 1989.
8. M. Pechoucek and V. Marík. Industrial deployment of multi-agent technologies: review and selected case studies. *Autonomous Agents and Multi-Agent Systems*, 17(3):397–431, 2008.
9. A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proc. of the First International Conference on Multiagent Systems (ICMAS)*, pages 312–319, 1995.
10. B. T. R. Savarimuthu, S. Cranefield, M. A. Purvis, and M. K. Purvis. Obligation norm identification in agent societies. *Journal of Artificial Societies and Social Simulation*, 13(4), 2010.
11. B. T. R. Savarimuthu, S. Cranefield, M. A. Purvis, and M. K. Purvis. Identifying prohibition norms in agent societies. *Artificial Intelligence and Law*, 21:1–46, 2012.
12. P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*, pages 380–394, 2004.
13. W. M. P. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
14. B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The ProM framework: A new era in process mining tool support. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN)*, pages 444–454, 2005.
15. H. H. Zhuo, D. H. Hu, C. Hogg, Q. Yang, and H. Muñoz-Avila. Learning HTN method preconditions and action models from partial observations. In *Proc. of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1804–1810, 2009.